The Signal Synchronous Multiclock Approach to the Design of Distributed Embedded System

Abdoulaye Gamatié, Thierry Gautier

The Signal Synchronous Multi-clock Approach to the Design of Distributed Embedded Systems

Abdoulaye Gamatié and Thierry Gautier

Abstract—This paper presents the design of distributed embedded systems using the synchronous multi-clock model of the SIGNAL language. It proposes a methodology that ensures a correct-by-construction functional implementation of these systems from high-level models. It shows the capability of the synchronous approach to apply formal techniques and tools that guarantee the reliability of the designed systems. Such a capability is necessary and highly worthy when dealing with safety-critical systems. The proposed methodology is demonstrated through a case study consisting of a simple avionic application, which aims to pragmatically help the reader to understand the manipulated formal concepts, and to apply them easily in order to solve system correctness issues encountered in practice. The application functionality is first modeled as well as its distribution on a generic hardware architecture. This relies on the endochrony and endo-isochrony properties of SIGNAL specifications, defined previously. The considered architectures include asynchronous communication mechanisms, which are also modeled in SIGNAL and proved to achieve message exchanges correctly. Furthermore, the synchronizability of the different parts in the resulting system is addressed after its deployment on a specific execution platform with multi-rate clocks. After all these steps, a distributed code can be automatically generated.

Index Terms—Distributed embedded systems, Correct-by-construction design methodology, Safety-critical domains, Formal validation, Synchronous model, Asynchronous mechanisms, Multi-clock, SIGNAL language

1 INTRODUCTION

There are various reasons to distribute embedded systems: the high performance enabled by the use of several computation units for a better response time, sometimes the geographical delocalization of the system elements, the replication of systems for fault tolerance, etc. Typical domains in which distributed embedded systems are useful are the following:

- *transportation*: the last generation of cars and aircraft increasingly combine computers and electrical linkages. This combination makes comfortable the control of the engine. This combination leads to so-called *fly-by-wire* systems, e.g., adopted in Toyota Prius automobile or in the A380.
- *telecommunication*: cell phone manufacturers and Internet highly face problems related to distributed and mobile embedded systems deployed on networks. A major issue is the compatibility between heterogeneous devices, and the robust and transparent communications in networks, while guaranteeing performance and real-time properties.
- *industrial automation*: it covers industrial robots and embedded mechatronic systems, which comprise technologies from engineering disciplines in computer software and hardware as well as in mechanics and automatic control. The issues concern the design of distributed architectures for scalable and reconfigurable mechatronic systems, and of embedded control.

The design of distributed embedded systems generally has to take into account several crucial aspects: the used communication mechanisms must be proved to be sufficiently robust, i.e. no loss of messages; for sub-systems located on different sites, the duration of communications must not alter the real-time properties of the overall system, the global functionality must be guaranteed, etc. So, the distribution of a system has to follow well-defined methodologies that offer the suitable means to clearly express its inherent concurrency and to validate its behavior w.r.t. functional and non functional requirements.

In the current study, we consider the *synchronous model* [10] for design and validation. This model offers an adequate formal basis to deal with the reliable design of embedded systems. Its basic assumption is that computation and communications are *instantaneous from the point of view of a logical time*, referred to as "*synchrony hypothesis*". This favors *deterministic* models of system behaviors for safe analysis.

1.1 The synchronous multi-clock model of SIGNAL

The design of distributed systems has been extensively studied for decades [44], [19]. The asynchrony [20] inherent to these systems appears *a priori* as an obstacle to their description with the synchronous model. According to [45], a fully synchronous system is characterized by the *boundness and knowledge* of: *i) processing speed*, *ii) message delivery delay*, *iii) local clock rate drift*, *iv) load pattern*, and *v) difference among local clocks*. A fully asynchronous system assumes none of these characteristics.

Distributed embedded systems are generally implemented with GALS architectures, which are suitably abstracted with the SIGNAL synchronous multi-clock or *polychronous* model [37]. The multi-clock model is gaining an increasing attention from academia and industry to address the modern electronic designs [13], [24]. According to the polychronous model, the activation *clock* of a system consists of the set of instants at which its components react. This set is only partially ordered since concurrent components may have their own (independent) activation clocks. When these components do not interact, it is not necessary that their clocks be synchronized. The central question is not to agree on a global time, but to agree on the common event occurrences during the interaction of components. Contrarily to the usual monolithic approach adopted by the other synchronous languages (e.g. LUSTRE, ESTEREL until recently), which consists of compact single-clocked designs, the

This work has been partly supported by the European project IST SAFEAIR (Advanced Design Tools for Aircraft Systems and Airborne Software) - Convention nº 1 00 C 0149 00 31307 00 5.

- *A. Gamatié is affiliated with CNRS/LIFL, Email: abdoulaye.gamatie@lifl.fr.*
- *T. Gautier is affiliated with INRIA, Email: thierry.gautier@irisa.fr.*

polychronous approach aims at designing in a modular fashion systems with multiple loosely coupled clocks so as to be able to deal with the complexity of their distribution.

Compared to the synchrony/asynchrony definition of [45], the polychronous model offers an intermediate vision: while it assumes the boundness of computation and communication activities, the difference among local activation clocks is *a priori* unknown. This vision is opposed to the monolithic one, which considers the existence of a unique common clock in a system, represented by a totally ordered set of instants. The SIGNAL language relies on the polychronous model.

FIG. 1 illustrates the design philosophy behind polychrony: to give an adequate abstract model of a system shown in Fig. 1(b), which enables a subsequent choice towards a distributed system implemented by *globally asynchronous locally synchronous* (GALS) architectures [17] shown in Fig. 1(a), or a single-clocked system implemented by centralized architectures shown in Fig. 1(c).

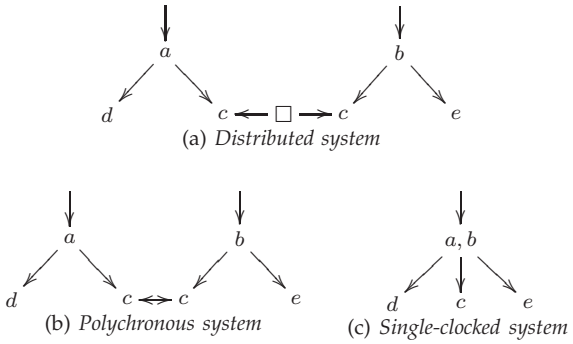


Fig. 1. Distributed, polychronous and single-clocked systems.

The polychronous system, depicted in Fig. 1(b), consists of two concurrent sub-systems that receive control from events a and b and conditionally communicate via an event c or do something else (d or e). It can be refined into a single-clocked system, Fig. 1(c), by synchronizing the events a and b. It can be also refined into a distributed system, Fig. 1(a), by distributing a and b on two different locations and by implementing c by a FIFO communication channel. In general, the reliability of such channels is implicitly assumed in the polychronous model. However, in this paper, it is explicitly proved.

1.2 Synchronous design of distributed systems

Among the large spectrum of literature devoted to the design of distributed systems [44], [19], we mainly concentrate below on related works that rely on the synchronous model.

As mentioned above, a major advantage of using such a model for the development of distributed embedded systems is that, on the one hand, the global correctness of systems can be trust-worthily guaranteed using the associated formal technology. It is particularly necessary when the system is used in a safety-critical domain such as automotive or avionics. This explains the numerous studies on program distribution problematics in synchronous languages. Most of these studies focus on *automatic program distribution* methods [27]: given a centralized synchronous program P and a distributed architecture A, the deployment of P on A is defined automatically, with the necessary inserted communication code, so that the resulting distributed program has the same functional behavior as P. However, note that beyond these studies, automatic program distribution has been widely investigated [32].

The major contributions on this topic in the synchronous approach community could be summarized according to the main synchronous languages [10]: ESTEREL, LUSTRE and SIGNAL.

In ESTEREL, we can mention the work of Berry and Sentovich [12] on the construction of GALS systems as synchronous circuits represented by a network of communicating *codesign finite state machines*. GALS architectures [17] consist of components that execute synchronously and communicate asynchronously. Another relevant work concerns the ESTEREL specification and programming of large-scale distributed real-time systems [33].

The LUSTRE language has been also used in several studies on the design of distributed systems. Girault [26] addressed the distribution of synchronous automata within the framework of this language. Afterwards, he focused on further issues such as automatic deduction of GALS systems from centralised synchronous circuits [28], and the optimized execution of desynchronized embedded reactive programs to guarantee real-time constraints [29]. We must note the very important achievements by Caspi and his colleagues on the same topics (see [27]). They recently proposed an approach to deploy LUSTRE programs on *time-triggered architectures* [16].

In SIGNAL, there have been lots of results on program distribution, mostly from the theoretical viewpoint. The earlier work of Chéron [18] dealt with the communication of separately compiled SIGNAL programs. Then, Maffeïs [39] showed how to abstract such programs into graphs in order to define the qualitative scheduling and partitioning of these graphs. The work of Aubry [6] focused on similar problems as Girault [26] by exploring the manual and semi-automatic distribution of synchronous dataflow programs in SIGNAL. While these studies were mostly devoted to the practical side, Benveniste and Le Guernic lead several theoretical works on the distribution of SIGNAL programs [8], [9], [25], [11], [37], [41]. The approach presented in this paper shows how these theoretical results are exploited in practice to define correct-by-construction models of distributed embedded systems.

Finally, we can notice other interesting contributions such as [30] in which Grandpierre showed, with Petri nets, how one can derive a distributed implementation from a synchronous dataflow specification, e.g. in LUSTRE or SIGNAL, of a given application while minimizing the response time w.r.t. real-time requirements. The SYNDEX environment [31] aims at providing designers with such program distribution facilities.

1.3 Our contribution: a seamless design methodology

While the above studies mainly concern automatic program distribution, in this paper we focus on how to design entirely distributed embedded systems with a synchronous language by proposing a general methodology, FIG. 2, composed of four steps detailed in the following chronological order:

1) *System specification and manual distribution*: modeling of application functionality, distributed hardware architecture, and their association. This step is similar to the initial step in usual harware-software codesign;
2) *Automatic transformations*: guaranteeing the global functional correctness of the distribution. These transformations exploit the endochrony and endo-isochrony properties of SIGNAL programs, presented in Section 2.2.2;
3) *Deployment on specific platforms*: instantiating the parts of the resulting model with components that represent

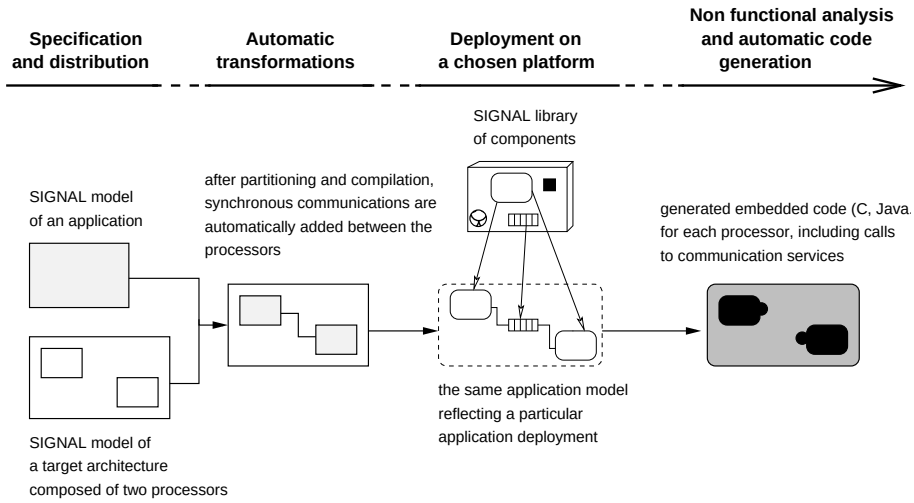


Fig. 2. Our design methodology for distributed embedded systems in SIGNAL.

specific platform mechanisms, e.g. for communication, synchronization. There are only few existing works devoted to the synchronous modeling of asynchronous mechanisms [34], [35], [23];

4) *Non functional analysis and automatic code generation*: checking the non functional (temporal) constraints induced by the chosen deployment and distributed code generation.

The above methodology is exposed through a case study. It is built on several complementary results obtained previously. The second step of this methodology uses an implementation of the theoretical results of [8], [9]. The third step relies on the insights gained from [21], [23] in which we have proposed a library of components consisting of synchronous models of asynchronous mechanisms for communication, synchronization or execution. Here, we consider the mechanism defined in [21] to describe the communications in the deployed system model of our case study. We also give the proof of queueing order preservation in this mechanism, which was an assumption previously. On the other hand, while the design approach presented in [23] mainly targets integrated modular avionics architectures, the current methodology aims at more general multi-processor architectures and fully makes use of the SIGNAL multi-clocked model. The fourth step uses the non functional analysis techniques proposed in [43], [36] and implements code generation approach defined in [25].

1.4 Example: a Flight Warning System

We consider a running example from the avionic domain, consisting of a Flight Warning System (FWS) to illustrate our methodology. This system is used in the Airbus A340 aircraft. It has been proposed by the Aerospatiale Company (France) as a case study in [38]. The FWS system is in charge of deciding on when and how to emit warning signals whenever there is an anomaly during the operational mode of an airplane. It is illustrated in FIG. 3 together with its implementation architecture. It consists of two cyclic concurrent processes:

- given an alarm a_i, the *alarm manager* process confirms a_i after a given period of time or removes a_i from the set of confirmed alarms depending on the fact that a_i is detected "present" or "absent";
- the *alarm notifier* process emits warning signals associated with confirmed alarms.

In this paper, the exact way alarms are made present or absent is supposed to be out of scope. This is achieved in another part of the aircraft.

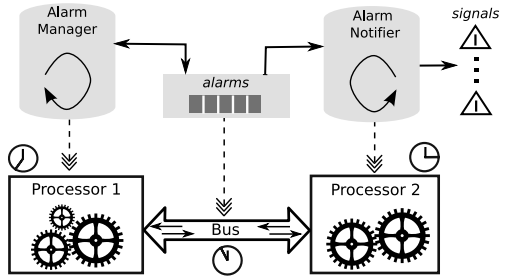


Fig. 3. Deployment of FWS on a platform.

Even though the above two processes interact, they execute independently. So, their activation clocks are *a priori* not strongly correlated. Such a system is particularly well-suited to be described and analyzed with the polychronous model. The specific choice of executing the processes separately on different processors is one possibility among others. Here, it allows us to illustrate the possible design issues resulting from the distribution of a system functionality and how to address them with our proposed methodology.

1.5 Outline

The remainder of this article is organized as follows: Section 2 introduces the synchronous language SIGNAL and its associated multi-clock semantic model. Then, the next sections present our methodology in detail through the design of FWS on a distributed platform. Section 3 is devoted to the first and second steps. It addresses the SIGNAL description of the FWS system, its associated hardware architecture and its correct distribution on the chosen architecture. Section 4 concentrates on the third step. It deals with the modeling of an asynchronous communication mechanism, represented by a FIFO queue that is proved to be correct w.r.t. its expected services. This mechanism is used for the deployment of the FWS system on a specific platform. Section 5 focuses on the last step of our methodology. It addresses clock synchronizability issues in the

deployed model of the FWS system and the automatic code generation. Finally, our proposition is discussed in Section 6 and concluding remarks are given in Section 7.

2 THE SIGNAL LANGUAGE

In order to define the formal semantics of SIGNAL, we introduce the polychronous semantic model in Section 2.1. Then, we give an overview of the language in Section 2.2.

2.1 The polychronous semantic model

We consider the following sets: $\mathcal{X}$ is a countable set of variables; $\mathcal{B} = \{ff, tt\}$ is a set of Boolean values where *ff* and *tt* respectively denote *false* and *true*; $\mathcal{V}$ is the domain of operands (at least $\mathcal{V} \supseteq \mathcal{B}$); and $\mathbb{T}$ is a dense set equipped with a partial order relation noted $\leq$, and with a greatest lower bound. The elements of $\mathbb{T}$ are called *tags*.

We now introduce the notion of *observation point*.

Definition 1 (observation points): A set of observation points is a set of tags $\mathcal{T}$ such that: $i)$ $\mathcal{T} \subset \mathbb{T}$; $ii)$ $\mathcal{T}$ *is countable*; $iii)$ *each pair of tags admits a lower bound in* $\mathcal{T}$.

The set $\mathcal{T}$ provides a discrete time dimension that corresponds to the logical instants at which the presence/absence of events can be observed during a system execution. The set $\mathbb{T}$ provides a continuous (physical) time dimension. So, the mapping of $\mathcal{T}$ on $\mathbb{T}$ allows one to move from "abstract" descriptions to "concrete" descriptions in the semantic model.

A *chain* $C \subseteq \mathcal{T}$ is a totally ordered set which admits a lower bound. The set of chains is denoted by $\mathcal{C}$. For a set of observation points $\mathcal{T}$, we denote by $\mathcal{C}_\mathcal{T}$ the set of all chains in $\mathcal{T}$. The notations $min(C)$ and $pred_C(t)$ respectively mean the minimum of C and the immediate predecessor of a tag $t \in C$.

Definition 2 (events, signals and behaviors, FIG. *4):*

- An *event* e on a given set of observation points $\mathcal{T}$ is a couple $(t, v) \in \mathcal{T} \times \mathcal{V}$.
- A *signal* on a given set of observation points $\mathcal{T}$ is a partial function $s \in \mathcal{T} \rightharpoonup \mathcal{V}$, associating values with observation points that belong to a chain $C \in \mathcal{C}_\mathcal{T}$. The set of signals on $\mathcal{T}$ is noted $\mathcal{S}_\mathcal{T}$. The domain of s is denoted by $tags(s)$.
- For a given set of observation points $\mathcal{T}$, a *behavior* b on $X \subseteq \mathcal{X}$ is a function $b \in X \rightarrow \mathcal{S}_\mathcal{T}$ that associates each variable $x \in X$ with a signal s on $\mathcal{T}$. We denote by $\mathcal{B}_{\mathcal{T},X}$ the set of behaviors of domain $X \subseteq \mathcal{X}$ on $\mathcal{T}$. The set $\mathcal{B}_\mathcal{T}$ represents the set that contains all the behaviors defined on the union of all the sets of variables on $\mathcal{T}$. Finally, we write $vars(b)$ and $tags(b) = \bigcup_{x \in vars(b)} tags(b(x))$ to respectively denote the domain of b and its associated set of tags. □

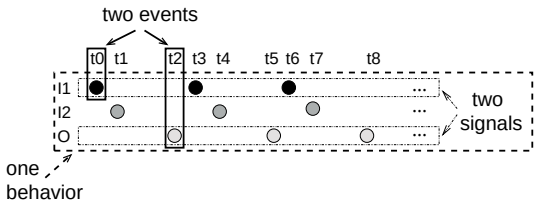


Fig. 4. Behavior, events and signals.

For any behavior b defined on $X \in \mathcal{X}$, we note $b_{|X'}$ its projection on a set of variables $X' \subset X$. In other words, $vars(b_{|X'}) = X'$ and $\forall x \in X', b_{|X'}(x) = b(x)$. The projection of b on the complementary of X' in X is denoted by $b_{/X'}$.

A signal can be seen, in fact, as an *elastic* with ordered *marks* (its tags). If such an elastic is *stretched*, the marks remain in the same order but new marks may be inserted between stretched marks. If the elastic gets unstretched, the marks become closer to each other, but they still remain in the same order. The same holds for a set of elastics, i.e., a behavior.

Definition 3 (stretching): For a given set of observation points $\mathcal{T}$, a behavior b_1 is less stretched than a behavior b_2, noted $b_1 \leq_{\mathcal{B}_\mathcal{T}} b_2$, iff there exists a bijection $f : tags(b_1) \rightarrow tags(b_2)$ following which b_1 and b_2 are isomorphic:
$\forall x \in vars(b_1),\quad f(tags(b_1(x))) = tags(b_2(x))$,
$\forall x \in vars(b_1)\ \forall t \in tags(b_1(x)),\quad b_1(x)(t) = b_2(x)(f(t))$,
$\forall t_1, t_2 \in tags(b_1),\quad t_1 \leq t_2 \Leftrightarrow f(t_1) \leq f(t_2)$,
and such that $\forall C \in \mathcal{C}_\mathcal{T},\quad \forall t \in C\ t \leq f(t)$. □

The stretching relation is a partial order on $\mathcal{B}_\mathcal{T}$. It induces an equivalence relation between behaviors.

Definition 4 (stretch-equivalence): For a given set of observation points $\mathcal{T}$, two behaviors b_1 and b_2 are stretch-equivalent, noted $b_1 \lesseqgtr b_2$, iff there exists a behavior b_3 less stretched than b_1 and b_2, i.e. $b_1 \lesseqgtr b_2$ iff $\exists b_3\ b_3 \leq_{\mathcal{B}_\mathcal{T}} b_1\ \ and\ \ b_3 \leq_{\mathcal{B}_\mathcal{T}} b_2$. □

The class of equivalence of a behavior following the stretch-relation forms a semi-lattice, which admits a minimal element. We call *strict* behaviors those which are minimal for the stretch-relation on $\mathcal{T}$. For a given behavior b, the set of all behaviors that are stretch-equivalent to b on $\mathcal{T}$ defines its *stretch-closure* on $\mathcal{T}$, noted b^*.

Definition 5 (stretch-closure of a behavior set): The stretch-closure of a set of behaviors p on a given set $\mathcal{T}$ of observation points is the set p^*, which includes all behaviors resulting from the stretch-closure of each behavior $b \in p$, i.e. $p^* = \bigcup_{b \in p} b^*$. □

The stretch-closure allows us to define the following *process* notion, which is a specific set of behaviors:

Definition 6 (process): For a given set of observation points $\mathcal{T}$, a process p is a stretch-closed set of behaviors, i.e. $p \in \mathcal{P}(\mathcal{B})$ such that $p = p^*$. □

We write $vars(p)$ to denote the set of variables of a process p (we say equivalently that p is defined on $vars(p)$). Every non empty process contains a subset $p_\downarrow \subseteq p$ of *strict* behaviors (for each $b_1 \in p$, there exists a unique $b_2 \in (p)_\downarrow$ such that $b_2 \lesseqgtr b_1$).

The notions presented in this section are sufficient to express the semantics of SIGNAL basic concepts within the polychronous model [37]. In the remainder of the paper, we also use them to characterize some model properties.

2.2 An overview of the language

2.2.1 Basics

SIGNAL [37], [14] is a dataflow relational language that handles unbounded series of typed values $(x_t)_{t \in \mathbb{N}}$, called *signals*, implicitly indexed by discrete time, and denoted as `x`. At any logical instant $t \in \mathbb{N}$, a signal may be present or absent: when present, it holds some value; when absent, it holds no value and is denoted by $\perp$ in the semantic notation. There is a particular type of signal called `event`. A signal of this type always holds the value *true* when it is present. The set of instants at which a signal `x` is present is referred to as its *clock*, noted $\hat{x}$. A *process* is a system of equations over signals that specifies relations between values and clocks of the signals. A *program* is a process. The whole SIGNAL language relies on the six primitive constructs, presented below, which define *elementary processes*. Unlike the LUSTRE and LUCID SYNCHRONE languages in which a program expresses a functional dataflow specification, in SIGNAL, a program implicitly expresses constraints on the involved signals, that must be satisfied by their

values on the one hand and by their clocks on the other hand (see TABLE. 1).

- *Instantaneous relations:* `y:= R(x1,...,xn)` where `y, x1,...,xn` are signals and `R` is a point-wise n-ary relation/function extended canonically to signals. This construct imposes `y, x1,...,xn` *i)* to be simultaneously present, and *ii)* to hold values satisfying the equation `y:= R(x1,...,xn)` whenever they occur. Its semantics is the following set of behaviors:

  $\{ \quad b \in \mathcal{B}_{|y,x_1,...,x_n} \mid tags(b(y)) = tags(b(x_1)) = ... = tags(b(x_n)) = C \in \mathcal{C}\backslash\emptyset \; \forall t \in C, b(y)(t) = [\![R]\!](b(x_1)(t), ..., b(x_n)(t)) \quad \}$

  where $[\![R]\!]$ is a point-wise interpretation of the relation `R`.

- *Delay:* `y:= x $ 1 init c` where `y, x` are signals and `c` is an initialization constant. This construct imposes *i)* `x` and `y` to be simultaneously present while *ii)* `y` must hold the value carried by `x` on its previous occurrence. Its semantics is the following set of behaviors:

  $\{0_{|x,y}\} \cup \{ \quad b \in \mathcal{B}_{|x,y} \mid tags(b(y)) = tags(b(x)) = C \in \mathcal{C}\backslash\emptyset, \; b(y)(min(C)) = c, \; \forall t \in C\backslash min(C), b(y)(t) = b(x)(pred_C(t)) \quad \}$

- *Undersampling:* `y:= x when c` where `y, x, c` are signals and `c` is of Boolean type. This construct imposes *i)* `y` to be present only when `x` is present and `c` holds the value *true*, while *ii)* `y` must hold the value carried by `x` at those logical instants. Its semantics is the following set of behaviors:

  $\{ \quad b \in \mathcal{B}_{|x,y,c} \mid tags(b(y)) = \{t \in tags(b(x)) \cap tags(b(c)) | b(c)(t) = tt\} \; \forall t \in tags(b(y)), b(y)(t) = b(x)(t) \quad \}$

- *Deterministic merging:* `z:= x default y` where `z, y, x` are signals. This construct imposes *i)* `z` to be present when either `x` or `y` are present while *ii)* `z` must always hold the value of `x` uppermost, otherwise it takes the value of `y`. Its semantics is the following set of behaviors:

  $\{ \quad b \in \mathcal{B}_{|s_1,s_2,s_3} \mid tags(b(s_3)) = tags(b(s_1)) \cup tags(b(s_2)) = C \in \mathcal{C} \; \forall t \in C, \; b(s_3)(t) = b(s_1)(t) \textbf{ if } t \in tags(b(s_1)) \textbf{ else } b(s_3)(t) = b(s_2)(t) \quad \}$

- *Composition of processes:* union of the equations defined in processes, leading to the conjunction of the constraints associated with these processes. For a given set of observation points $\mathcal{T}$, the composition on $\mathcal{T}$ of processes p_1 and p_2, noted $p_1 \mid p_2$, is a process $p = (\{b \mid b_{|vars(p_1)} \in p_1, b_{|vars(p_2)} \in p_2\})^*$ **and** $vars(p) = vars(p_1) \cup vars(p_2)$.

- *Restriction (or Hiding):* local declarations in a process. The restriction, noted `p where x` (or `p/x` for short), of a process p defined on $X \subseteq \mathcal{X}$ to a process defined on $X\backslash\{x\}$, is defined as the following set of behaviors: $(\{b_2 \mid \exists b_1 \in p \wedge b_2 = b_{1/\{x\}}\})^*$.

These constructs are expressive enough to derive new constructs of the language for comfort and structuring.

2.2.2 Clock calculus, endochrony and endo-isochrony

In SIGNAL, clocks are fundamentally the main means to express control-related properties, i.e. synchronizations between signals. They are ordered according to the following relation: a clock κ_1 is said to be *greater than* a clock κ_2, which is denoted by $\kappa_1 \geq \kappa_2$, if κ_2 is included in κ_1 in terms of sets of instants. The set of clocks associated with this relation is a lattice.

Clock calculus. The purpose of the *clock calculus* [37] is, on the one hand, to determine the existence of a greatest clock, called *master clock*, from which all clocks of a program can be extracted, and on the other hand, to verify the consistency of synchronization relations between signals. Nonetheless, in some programs, such a unique master clock may not exist.

TABLE 1
Clock constraints in primitive constructs.

constructs	clock constraints
`y := f(x1,...,xn)`	$\hat{y} = \hat{x}_1 = ... = \hat{x}_n$
`y := x $1 init c`	$\hat{y} = \hat{x}$
`y := x when b`	$\hat{y} = \hat{x} \cap [b]$, $[b] \cup [\neg b] = \hat{b}$ and $[b] \cap [\neg b] = \emptyset$
`z := x default y`	$\hat{z} = \hat{x} \cup \hat{y}$
p_1 `|` p_2	$constraints(p_1) \cap constraints(p_2)$
`p where x`	$constraints(p)$

In this case, there are several local master clocks (see Section 3.1.2). TABLE. 1 gives the clock properties associated with primitive constructs of SIGNAL (here, $[b]$ – resp. $[\neg b]$ – denotes the set of instants where the Boolean expression `b` is present and *true* – resp. *false* –). Such properties are automatically inferred by the compiler from any program to be analyzed. A specific option of the compiler enables to generate a file containing these properties under the form of clock relations defined in a SIGNAL program. Clock hierarchies can therefore be extracted from such a program.

SIGNAL allows one to explicitly manipulate clocks through some derived constructs such as the following ones (the equivalent expressions are given with primitive operators):

- *Sampling.* `y:= when b` $\stackrel{def}{\equiv}$ `y := b when b`. It expresses the fact that `event` signal `y` occurs whenever the signal `b` is *true*.
- *Clock union.* `y:= x1 ^+ x2` $\stackrel{def}{\equiv}$ `y := ^x1 default ^x2`. It denotes the set of instants at which at least one signal `xi` occurs. Clock *intersection* and *difference* are also defined.
- *Synchronizer.* `x1 ^= x2` $\stackrel{def}{\equiv}$ `(| x := ^x1 = ^x2 |)/x`. It means that `x1` and `x2` have the same clock or are *synchronous*.

Endochrony and endo-isochrony: key properties for program distribution. The safe distribution of SIGNAL programs is possible thanks to two fundamental notions, *endochrony* and *endo-isochrony*, which have been formally defined in [37], [9]. In this paper, we only recall them informally and put them to work on our case study introduced in Section 1.4.

An endochronous program p is a program that can be executed in an environment, which only provides p with the values of its inputs, without any information about their status (present or absent). p is able to reconstruct a unique synchronous behavior from any external (asynchronous) input flow of values. It is insensitive to external propagation delays. From the viewpoint of clocks, such a program holds a master clock κ that plays the role of its activation clock. All the other clocks of p are necessarily defined recursively from κ. They are smaller than κ following the partial order $\geq$. The global clock hierarchy in p therefore forms a *tree* where the root node represents the master clock of p (see Section 3.1.1). Given any SIGNAL program `p`, the compiler is able to check whether or not `p` is endochronous. An endochronous program is *deterministic*: for the same functional input values it yields the same functional output values whatever the external propagation delays are. The elementary processes defined by the instantaneous relations and delay primitive constructs are examples of very simple endochronous programs. All signals involved in these processes have the same clock, which is also the master clock. On the contrary, the elementary processes defined by the undersampling and merging primitive constructs are not endochronous programs since there is no master clock from which the clocks of involved signals can be inferred in

such a process. A more complex example of non endochronous program is shown in Section 3.1.2.

The *isochrony* property [9] is used as a criterion under which the synchronous composition of a pair of processes p_1 and p_2 is equivalent to their asynchronous composition (i.e. involving "*send/receive*" like communications). It means that the order of the messages exchanged synchronously by p_1 and p_2 is always preserved when exchanged asynchronously. Thus, p_1 and p_2 can be safely deployed on GALS architectures. So, isochrony is an interesting property when one needs to describe asynchronous behavior while using the synchronous (multi-clock) model. A very similar theoretical result can be found in [41], where the authors address the functional equivalence of asynchronous composition and synchronous composition with bounded and unbounded FIFO channels, within the polychronous model. In our methodology, we rather consider a more constructive version of isochrony, called *Endo-isochrony*.

Endo-isochrony [37] requires that both p_1 and p_2 are endochronous each, as illustrated in Section 3.1.2; and imposes to their communicating part (i.e. the process defined by the restriction on their common variables) to be endochronous as well. Endo-isochrony is a sufficient criterion but not necessary to have a pair of isochronous processes. It is a stronger criterion. However, its advantage is that it permits to deal with the isochrony property of p_1 and p_2 using the compiler by checking the endochrony of the different sub-parts of the composition, hence making things more pragmatic. Endo-isochrony is illustrated in Section 3.1.2.

2.2.3 A rich tool-set for validation

The SIGNAL design environment, POLYCHRONY [14], offers a compiler and a model-checker that support the trustworthy and pragmatical design approaches for safety-critical systems.

Two kinds of functional properties are distinguished about SIGNAL programs: *invariant* properties, e.g. consistency of clock constraints, endochrony of a program, and *dynamical* properties, e.g., reachability, liveness (see Section 4.2). The compiler itself statically addresses invariant properties. Dynamical properties are addressed by the SIGALI model checker [40], which relies on the theory of polynomial dynamical systems. Roughly speaking, a SIGNAL program is abstracted into a system of polynomial equations representing a symbolic automaton over the set $\mathbb{Z}/3\mathbb{Z} = \{-1, 0, 1\}$. This encodes all the possible status of any Boolean signal: 1 for *true*, -1 for *false*, and 0 for $\perp$. For a non-Boolean signal, only the fact that this signal is *present* (whatever its value is) or *absent* is encoded. The presence is denoted by 1, and the absence by 0. It must be noted that this "translation" *fully takes into account information about Boolean variables (values and clocks), whereas for non-Boolean signals, information on values are lost*. Therefore, it is important that a SIGNAL program that will be analyzed by SIGALI is specified as much as possible using Boolean variables, since reasoning capabilities capture only synchronization and logic properties.

Finding a Boolean abstraction for a program is always possible. However, this abstraction may sometimes be not meaningful enough to adequately capture the properties of interest. Typically, some numerical properties require more powerful abstraction domains such as intervals [22] or polyhedra (which may lead to complex analyses). These possible extensions have been studied previously in the SIGNAL compiler in order to improve its analysis capability. Nevertheless, with SIGALI, the $\mathbb{Z}/3\mathbb{Z}$ ternary abstract encoding is necessary. In general, a good way to obtain analyzable programs in this context consists in defining Boolean signals, usable for characterizing the relevant properties. In most cases, when such properties concern the control of a program behavior, i.e. synchronization constraints, one can easily find a direct Boolean encoding.

3 SPECIFICATION AND CORRECT DISTRIBUTION

3.1 Model of the FWS functionality

Given an application, its functionality is first described as a hierarchical SIGNAL process $p = p_1 \mid ... \mid p_n$. Here, p_1 and p_2 denote the alarm notifier and manager processes (Section 3.1.1) while p represents the FWS system (Section 3.1.2).

3.1.1 Alarm manager and notifier processes

FIG. 5 gives an excerpt of the models of the processes composing FWS. Both are defined in a very similar way. So, we only present one of them: `Alarm_Manager`. The interface signals `alarm_in` and `alarm_out`, at lines 3 and 4, are both declared as arrays of alarms. In SIGNAL, the input and output signals are introduced by the symbols "?" and "!" respectively. Here, their dimension is some fixed integer constant `n`. The type of an alarm, called `alarm_type`, is a structured type with two fields: `pres` (resp. `conf`) of Boolean type that is *true* when an alarm is "present" (resp. "confirmed") and *false* when an alarm is "absent" (resp. "removed").

The body of the process is composed of five equations, from line 5 to line 11. The local signals `cnt`, `zcnt` and `start_confirm` are used to specify when the process confirms alarms in equations at lines 5 to 8. The signal `cnt` plays the role of a *counter* of logical instants. It is set to `k-1` whenever its previous value `zcnt` becomes zero; otherwise its previous value is decremented by one. The static parameter `k` at line 2 can be somehow seen as an initialization period value of `cnt`. The signal `start_confirm` denotes the logical instants at which a confirmation begins, described by equations at lines 8 and 9. It occurs when the counter value reaches some amount of time denoted by `delay` within each cycle.

Note that the sub-process `Alarm_Confirm`, defined at line 15 and invoked at line 10, to produce the output `alarm_out`, uses a derived construct of SIGNAL, referred to as *array of processes*, that enables to describe instantaneous iterations [14].

From the above equations and according to TABLE. 1, we can deduce the following system of clock constraints associated with the process `Alarm_Manager` (for a signal `s`, we denote by *clk_s* its associated clock):

$$\begin{aligned} clk\_cnt &= [zcnt = 0] \cup clk\_zcnt && \text{(lines 5 \& 6)} \\ clk\_zcnt &= clk\_cnt && \text{(line 7)} \\ clk\_start\_confirm &= [zcnt = delay] && \text{(line 8)} \\ clk\_alarm\_in &= clk\_start\_confirm && \text{(line 9)} \\ clk\_alarm\_out &= clk\_alarm\_in && \text{(lines 10 \& 19)} \end{aligned} \quad (1)$$

After reductions of the above system (1), we obtain the following clock hierarchy:

$$clk\_cnt \equiv clk\_zcnt$$

$$[zcnt \neq delay] \qquad [zcnt = delay] \equiv clk\_start\_confirm \equiv clk\_alarm\_in \equiv clk\_alarm\_out$$

We can see that the master clock is the one of `cnt`, which is the same for `zcnt`. The other clocks of the process are defined following a partitioning of the values of `zcnt` w.r.t. the value of `delay`. As a result, the process `Alarm_Manager` is

```
1   process Alarm_Manager =                    process Alarm_Notifier =
2    { integer k,delay }                        { integer k,delay }
3    ( ? [n] alarm_type alarm_in;               ( ? [n] alarm_type alarm;
4      ! [n] alarm_type alarm_out; )              ! event s_0,...,s_n-1; )
5    (| cnt:= ((k-1) when (zcnt = 0))           (| cnt:= ((k-1) when (zcnt = 0))
6                  default (zcnt - 1)                         default (zcnt - 1)
7     | zcnt:= cnt $ init 0                      | zcnt:= cnt $ init 0
8     | start_confirm:= when(zcnt = delay)       | start_notif:= when (zcnt = delay)
9     | alarm_in ^= start_confirm                | alarm ^= start_notif
10    | alarm_out:= Alarm_Confirm(alarm_in)       | (s_0,...,s_n-1):= Alarm_Notif(alarm)
11    |)                                         |)
12  where                                      where
13    integer cnt, zcnt;                         integer cnt, zcnt;
14    event start_confirm;                       event start_notif;
15    process Alarm_Confirm =                    process Alarm_Notif =
16     ( ? [n] alarm_type in;                     ( ? [n] alarm_type alarm;
17       ! [n] alarm_type out; )                    ! event s_0,...,s_n-1; )
18     (| array i to n-1 of                       (| s0:= when alarm[0].conf
19          out[i].conf := in[i].pres              | ...
20        end                                      | s_n-1:= when alarm[n-1].conf
21      |);                                        |);
22  end;                                       end;
```

Fig. 5. Excerpt of the SIGNAL code of alarm manager (left) and notifier processes (right).

endochronous (hence deterministic). In a similar way, we prove that the process `Alarm_Notifier` is also endochronous. More generally, the SIGNAL compiler allows one to automatically check the endochrony property of a process based on this technique and to generate an associated simulation code.

3.1.2 Global model of the FWS system

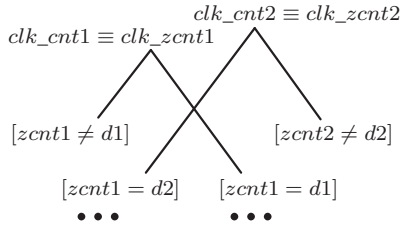
```
1   process FW_System =
2    { integer k1,k2,d1,d2 }
3    ( ? [n] alarm_type alarm;
4      ! event s_0,...,s_n-1; )
5    (| tmp:= Alarm_Manager{k1,d1}(alarm)
6     | (s_0,...,s_n-1):=
7                 Alarm_Notifier{k2,d2}(tmp)
8     |)
9    where
10    [n] alarm_type tmp;
11    process Alarm_Manager = ...;
12    process Alarm_Notifier = ...;
13   end;
```

Fig. 6. Excerpt of the SIGNAL code of FWS.

The process depicted by FIG.6 represents the FWS model. It takes as input a collection of alarms represented by `alarm` (line 3), and produces as outputs the signals denoted by `s_0`, ..., `s_n-1`. These outputs are generated by `Alarm_Notifier`.

The equations from lines 5 to 7 define the interaction between the concurrent processes in the FWS system: the input of process `Alarm_Notifier` are the output of process `Alarm_Manager`. This transfer of data is realized via the local signal `tmp`. It means that the two processes must agree on the set of specific instants at which this transfer is possible.

Let us analyze the clock properties induced by the definition of process `FW_System`. We obtain the following two-rooted clock hierarchy:



We clearly distinguish the clock trees associated with `Alarm_Manager` and `Alarm_Notifier`. There is no master clock because the clocks of the sub-processes (`clk_cnt1` for `Alarm_Manager` and `clk_cnt2` for `Alarm_Notifier`) are included in none of each other. However, they share common instants at which the alarm transfer is done between the manager and the notifier processes. Such instants are defined only when both `zcnt` in `Alarm_Manager` and `zcnt` in `Alarm_Notifier` are respectively equal[1] to `d1` and `d2`. Such a requirement appears under the form of a clock constraint generated by the SIGNAL compiler. The corresponding code which is automatically produced ("`- force`" option of the compiler) from such a model contains exception messages that are thrown whenever the clock constraints are not satisfied during code execution.

In such a situation, it means that either the environment of the two concurrent processes is able to guarantee that they can synchronize at the required specific logical instants, and the model will execute as expected; or it is not possible, and the output of the whole system model is undefined whenever the clocks of the two processes fail to agree. In SIGNAL, the special construct called `assert` offers the way to define assumptions in programs [14]. It could be used here to meet the requirement on the concurrent execution of `Alarm_Manager` and `Alarm_Notifier` processes.

On the other hand, one can think of another specification of the `FW_System` model in which the confirmed alarm values `tmp` are transferred via a "clock-less" memory, which is made available whenever a process needs to make an access to this memory. This is obtained very easily by introducing a new equation in the body of `FW_System`, which defines a local signal `tmp_mem` as follows: `tmp_mem := (var tmp)`. Then, instead of putting `tmp` as input for `Alarm_Notifier` at line 7 in FIG. 6, we consider `tmp_mem`. The `var` operator allows one to memorize the value of `tmp` in `tmp_mem` [14]. The clock of `tmp_mem` is the one of the context in which it is used, i.e., the memorized value is available whenever required.

In this second version of the `FW_System` model, there is

1. Note that `zcnt` in `Alarm_Manager` and `zcnt` in `Alarm_Notifier` are different instances; they have different clocks and evolve according to two different logical time scales.

no longer clock constraints on the specific instants at which the concurrent processes must exchange information. Their associated master clocks are completely independent, with a possible empty set of common instants. In this case, the compiler generates no clock constraints and automatically produces a code without any exception message.

A very interesting feature of the `FW_System` model is that it satisfies the endo-isochrony property: first, `Alarm_Manager` and `Alarm_Notifier` models are both endochronous; second, their communicating part which consists of the single signal `tmp` (see FIG. 6) is trivially endochronous. Indeed, a single signal yields a clock hierarchy with only one clock node, which consequently forms a clock tree. The clock hierarchy associated with `FW_System` typically illustrates the endo-isochrony of a pair of processes. In particular, the synchronous communication between `Alarm_Manager` and `Alarm_Notifier` via `tmp` can be equivalently replaced by an asynchronous communication while still preserving the semantics of the `FW_System` model.

At this stage, some preliminary verification can be done on the `FW_System` model in order to make sure that it conforms to its expected specification requirements. Typically, one may check the absence of null clocks in the defined model using the compiler. The presence of such clocks denote a possible absence of reactions during the execution of the program. This is often undesirable for reactive systems.

3.2 Architecture model and manual distribution

The hardware architecture on which the previous application functionality model will be mapped is now defined. For the sake of simplicity, only processors are represented. Concretely, each processor is graphically modeled as an empty box.

Afterwards, one can proceed to the mapping of the application functionality on the hardware architecture in the POLYCHRONY graphical user interface. A SIGNAL code can be duplicated in different processor boxes; so, code replication is allowed. From this mapping, a new description of the system denoted by $p' = p'_1 \mid ... \mid p'_m$ is obtained, which reflects the target hardware architecture (m denotes the number of processors). FIG. 7 illustrates the distribution of the FWS model on a two-processors architecture, i.e. $m = 2$.

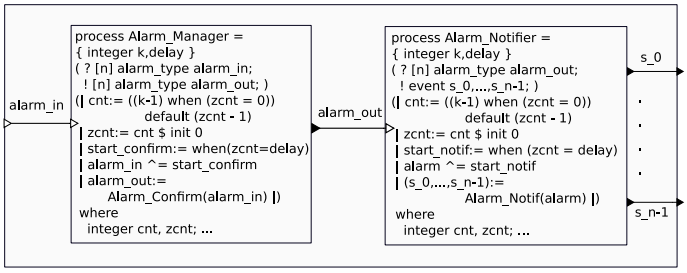


Fig. 7. Graphical distributed model of the FWS system.

We can notice that the sub-processes p'_j may not ne necessarily endochronous: after the mapping, p'_j can be obtained from the composition of different sub-processes p_i characterized by different clock trees without a common root clock.

3.3 Automatic transformations

Let us consider the model p' resulting from the mapping of the application functionality on the processors. The main purpose is to transform the process p' so that: *i)* p' fully preserves the functional semantics of p and *ii)* the sub-processes p'_j become endochronous and p' itself becomes endo-isochronous.

So, each process p'_j is *compiled modularly in the context of* p: from the clock hierarchy of p, the clock hierarchy of p'_j is transformed in such a way that it becomes a clock tree, i.e. p'_j becomes endochronous. For that, Boolean interface signals are used (each root node in a clock hierarchy of a non endochronous process is associated with a Boolean signal, and all these signals are made synchronous). On the other hand, some transformations are performed in order to render the communication part endochronous between the endochronous processes p'_j. For that, additional communicating signals are inserted between processes, which enable each process p'_j to receive all information it requires to execute. The result of these transformations is carried out automatically on the graphical representation of p' within the POLYCHRONY editor. In the obtained transformed process, denoted by p'', the information exchanges between processes are described by instantaneous communications. Here again, possible verification techniques can be applied to p'' in order to ensure that it still preserves the functional properties of p.

For our FWS system model, all the above transformations are not necessarily required since `FW_System` is already endo-isochronous as shown before. The only effect of these transformations is the automatic insertion of the communicating ports on processor boxes and the connections between these ports.

4 DEPLOYMENT ON SPECIFIC PLATFORMS

In order to refine the descriptions obtained from the previous steps of the methodology, a set of predefined components is proposed, which is usable to instantiate various aspects of the system: asynchronous communication mechanisms such as the FIFO buffer modeled in this paper, execution supports (e.g., tasks and processes), system-level primitives for task management as well as for synchronization and inter-task communication services (e.g. the APEX service models [23]). These components have been modeled in SIGNAL and are grouped in a library available in the POLYCHRONY platform.

4.1 Design of a library of components

We consider the FIFO queue presented in [21], to show how asynchronous communication mechanisms can be defined in SIGNAL, and proved to be correct. This queue is well adapted as mechanism to model a message buffer on which send/receive requests can be achieved by endo-isochronous processes. In addition, in the architecture level, it will be easily extended to model a bus. Compared to [21], here we give a more complete presentation of the design and analysis. A first model of FIFO queue is presented (`basic_ FIFO`); then, a more constrained version (`safe_ FIFO`) is defined from the first one and its key properties are checked.

4.1.1 A basic FIFO queue

We call *basic* FIFO, a message queue that works as follows.

On a *write request*, the incoming message is inserted in the queue regardless of its size limit. When the queue was previously full, the oldest message is lost. The other messages are shifted forward, and the incoming one is inserted in the queue. On a *read request*, there is an outgoing message whatever the queue status is. When it was previously empty, two situations are distinguished: if there is not yet any written message,

```
1   process basic_FIFO =
2   { type msg_type; integer size; msg_type def_msg; }
3   ( ? msg_type msg_in; event access_clock;
4     ! msg_type msg_out; integer nbmsg; boolean OK_write, OK_read; )
5   (| nbmsg := ((prev_nbmsg + 1) when (^msg_in) when OK_write)
6                default ((prev_nbmsg-1) when (^msg_out) when OK_read) default prev_nbmsg
7    | prev_nbmsg := nbmsg$1 init 0
8    | OK_write := prev_nbmsg<size
9    | OK_read := prev_nbmsg>0
10   | queue := (msg_in window size) cell (^access_clock)
11   | msg_out := prev_msg_out when (not OK_read) when (^msg_out)
12                default queue[size - prev_nbmsg] when (^msg_out)
13   | prev_msg_out := msg_out $ 1 init def_msg
14   | nbmsg ^= access_clock
15   |)
16  where
17   integer prev_nbmsg; [size]msg_type queue; msg_type prev_msg_out;
18  end;
```

msg_in	:	$\perp$	4	6	$\perp$	$\perp$	$\perp$	5	7	8	$\perp$	$\perp$	...
access_clock	:	t	t	t	t	t	t	t	t	t	t	t	...
msg_out	:	-1	$\perp$	$\perp$	4	6	6	$\perp$	$\perp$	$\perp$	7	8	...
nbmsg	:	0	1	2	1	0	0	1	2	2	1	0	...
OK_write	:	t	t	t	f	t	t	t	t	f	f	t	...
OK_read	:	f	f	t	t	t	f	f	t	t	t	t	...

Fig. 8. Model of the basic FIFO and an execution trace (`msg_type, size` and `def_msg` are respectively `integer`, 2 and -1).

an arbitrary message called *default message* is returned; else the outgoing message is the message that has been read last. Furthermore, for simplicity we suppose that simultaneous write/read requests on the queue never occur.

The SIGNAL model corresponding to the basic FIFO is given in FIG. 8. It is defined as a SIGNAL process, represented by the keyword `process` (line 1). The static parameters `msg_type`, `size` and `def_msg`, declared at line 2, respectively denote the type of messages, the size limit of the queue, and the default message value. The input signals `msg_in` and `access_clock` (line 3), are respectively the incoming message (its presence denotes a write request), and the queue access clock (i.e. instants of read/write requests). The output signals (line 4), are `msg_out`, `nbmsg`, `OK_write` and `OK_read`. They respectively represent the outgoing message, the current number of messages in the queue, and conditions for writing and reading.

Now, we can take a look at the meaning of the statements in the process body. Let us begin with the equation at line 7; it defines the local signal `prev_nbmsg`, which denotes the previous number of messages in the queue. This signal is used in equations at lines 8 and 9 to define respectively when the queue can be "safely" written (the size limit is not reached), and read (there is at least one message received). This is the meaning of the signals `OK_write` and `OK_read`.

The statement at lines 5 and 6 expresses how the current number of messages is calculated. The previous value of this number is incremented by one on a write request when the queue was not full. It is decremented by one on a read request when the queue was not empty. Otherwise, it remains unchanged. The equation at line 14 states that the value of `nbmsg` is defined whenever there is a request on the queue.

The equation at line 10 defines the message queue. The signal `queue` is an array of dimension `size` that contains the `size` latest values of `msg_in` (expressed by the `window` operator). The `cell` operator makes the signal `queue` available when `access_clock` is present. The semantics of these derived constructs is as follows:

- *Sliding windows.* In `y:= x window k init y_init`, the signal `y` is an array of size $k \geq 1$ whose elements have the same type as signal `x`; and `y_init` is an array of dimension $k' \geq k-1$, containing initialization values. This equation defines a sliding windows on `x`, of constant size `k` such that:
$$(\forall t \geq 0) \quad \big((t+i \geq k) \Rightarrow (y_t[i] = x_{t-k+i+1}) \vee (1 \leq t+i < k) \Rightarrow (y_t[i] = y\_init[t-k+i+2])\big)$$
- *Memory.* The expression `y := x cell b init c` enables to *memorize* the values of the signal `x`. It is defined as:
```
(| y := x default (y $ 1 init c)
 | y ^= x ^+ (when b) |)
```
The signal `y` takes the value of `x` when `x` is present. Otherwise, when `x` is absent and the signal `b` is present and holds the value *true*, `y` memorizes the latest value of `x`. When `b` is present and *true* before the first occurrence of `x`, `y` is initialized with the constant `c`. The clock of `y` is the union of $\hat{x}$ and $[b]$.

Finally, the statement at lines 11 and 12 means that on a read request (i.e. at the clock `^msg_out`), the outgoing message is either the previous one if the FIFO is empty (defined at line 13), or the oldest message in the queue.

4.1.2 A safe FIFO queue

In the model depicted in FIG. 9, the interface is slightly different from that of `basic_FIFO`. The new input signal `get_mess` denotes a read request. The signal `nbmsg` is local.

The statement at line 9 defines the access clock as the union of instants at which read/write requests occur. Equations at lines 10 and 11 specify the safe access to the queue in `basic_FIFO`. The process call at lines 12, 13 and 14 has the local signal `new_msg_in` as input. This signal is defined only when `basic_FIFO` was not full (line 10). The equation at line 11 expresses that on a read request, a message is enqueued only when `basic_FIFO` was not empty. In the trace in FIG. 9, the same parameters as for `basic_FIFO` are considered.

Through the above modeling approach, we observe that modularity and reusability are key features of the SIGNAL programming. They favor component-based design.

FIG. 10 shows the FWS model in which the communication between `Alarm_Manager` and `Alarm_Notifier` is achieved via the `safe_FIFO` model. Note the modification in the `Alarm_Notifier` at line 27 where the new output signal `get_msg` denotes read requests on `safe_FIFO`.

```
1   process safe_FIFO =
2   { type msg_type;
3     integer size;
4     msg_type def_msg; }
5   ( ? msg_type msg_in;
6       event get_msg;
7     ! msg_type msg_out;
8       boolean OK_write, OK_read; )
9   (| access_clock := msg_in ^+ get_msg
10   | new_msg_in := msg_in when OK_write
11   | msg_out ^= get_msg when OK_read
12   | (msg_out,nbmsg,OK_write,OK_read):=
13         basic_FIFO{msg_type,size,def_msg}
14           (new_msg_in,access_clock)
15   |)
16  where
17   use basic_FIFO; event access_clock;
18   integer nbmsg; msg_type new_msg_in;
19  end;
```

msg_in	:	⊥	4	6	⊥	⊥	⊥	5	7	⊥	...
get_msg	:	*t*	⊥	⊥	*t*	*t*	*t*	⊥	⊥	*t*	...
msg_out	:	⊥	⊥	⊥	4	6	⊥	⊥	⊥	5	...
OK_write	:	*t*	*t*	*t*	*f*	*t*	*t*	*t*	*t*	*f*	...
OK_read	:	*f*	*f*	*t*	*t*	*t*	*f*	*f*	*t*	*t*	...

Fig. 9. Model of the safe FIFO with an execution trace.

```
1   process FW_System =
2    { integer k1,k2,d1,d2 }
3    ( ? [n] alarm_type alarm;
4      ! event s_0,...,s_n-1; )
5    (| tmp:= Alarm_Manager{k1,d1}(alarm)
6     | (alarm_conf,OK_write,OK_read):=
7         safe_FIFO{[n] alarm_type,s,a_init}
8          (tmp, get_msg)
9     | (get_msg, s_0,...,s_n-1):=
10       Alarm_Notifier{k2,d2}(alarm_conf)
11    |)
12  where
13   use safe_FIFO; event get_msg;
14   [n] alarm_type tmp,alarm_conf;
15   boolean OK_write, OK_read;
16   constant integer s =...;
17   constant [n] alarm_type a_init =...;
18   process Alarm_Manager = ...;
19   process Alarm_Notifier =
20    { integer k,delay }
21    ( ? [n] alarm_type alarm;
22      ! event get_msg, s_0,...,s_n-1; )
23    (|cnt:= ((k-1) when (zcnt = 0))
24               default (zcnt - 1)
25     |zcnt:= cnt $ init 0
26     |start_notif:= when (zcnt = delay)
27     |get_msg := when start_notif
28     |(s_0,...,s_n-1):= Alarm_Notif(alarm)
29     |)
30   where
31    integer cnt, zcnt; event start_notif;
32    process Alarm_Notif =
33     ( ? [n] alarm_type alarm;
34       ! event s_0,...,s_n-1; )
35     (| s0:= when alarm[0].conf
36      | ...
37      | s_n-1:= when alarm[n-1].conf
38      |);
39   end;
40  end;
```

Fig. 10. Excerpt of the FWS model including `safe_FIFO`.

4.2 Ensuring the reliability of components

4.2.1 Safety and fairness

To check the properties of the `safe_FIFO` process, we consider an abstraction based on its *state variables*, i.e. signals defined by *delay* or *memory* operators in the process. These signals feature the dynamics of the system defined by the process. Here, the state variables are `nbmsg`, `queue` and `prev_msg_out` (defined in `basic_FIFO`). They entirely reflect the dynamics of the equation system associated with `safe_FIFO`. Let us represent the state of this system by σ. When the queue is full (resp. empty), the value of σ is noted $full$ (resp. $empty$); otherwise it is equal to $none$. Let $[\![\texttt{safe\_FIFO}]\!]$ denote the set of behaviors associated with `safe_FIFO` and b a behavior that belongs to this set, we address the following important properties of the `safe_FIFO` in order to guarantee its robustness:

Property 1 (safe read requests): The `safe_FIFO` queue never delivers an output message on a read request whenever it is empty (and its state remains unchanged): $\forall b \in [\![\texttt{safe\_fifo}]\!]$, $\mathcal{T} = tags(b(get\_msg))$,

$$\begin{aligned} &t \in \mathcal{T} \wedge t \neq min(\mathcal{T}) \wedge b(\sigma)(pred_{\mathcal{T}}(t)) = empty \\ &\quad \Rightarrow b(\sigma)(t) = b(\sigma)(pred_{\mathcal{T}}(t)) = empty \wedge \\ &\qquad t \notin tags(b(msg\_out)) \end{aligned} \tag{2}$$

Property 2 (safe write requests): The `safe_FIFO` queue never accepts an input message on a write request whenever it is full (and its state remains unchanged): $\forall b \in [\![\texttt{safe\_fifo}]\!]$, $\mathcal{T} = tags(b(msg\_in))$,

$$\begin{aligned} &t \in \mathcal{T} \wedge t \neq min(\mathcal{T}) \wedge b(\sigma)(pred_{\mathcal{T}}(t)) = full \\ &\quad \Rightarrow b(\sigma)(t) = b(\sigma)(pred_{\mathcal{T}}(t)) = full \wedge \\ &\qquad t \notin tags(b(new\_msg\_in)) \end{aligned} \tag{3}$$

We recall that the signal `new_msg_in`, which is defined in the `safe_FIFO` process, at line 10, denotes incoming messages that are effectively enqueued.

Property 3 (fair read requests): A message can always be read from the `safe_FIFO` queue, whenever it is not empty: $\forall b \in [\![\texttt{safe\_fifo}]\!]$, $\mathcal{T} = tags(b(get\_msg))$,

$$\begin{aligned} &t \in \mathcal{T} \wedge t \neq min(\mathcal{T}) \wedge b(\sigma)(pred_{\mathcal{T}}(t)) \neq empty \\ &\quad \Rightarrow t \in tags(b(msg\_out)) \end{aligned} \tag{4}$$

Property 4 (fair write requests): A message can always be written in the `safe_FIFO` queue, whenever it is not full: $\forall b \in [\![\texttt{safe\_fifo}]\!]$, $\mathcal{T} = tags(b(msg\_in))$,

$$\begin{aligned} &t \in \mathcal{T} \wedge t \neq min(\mathcal{T}) \wedge b(\sigma)(pred_{\mathcal{T}}(t)) \neq full \\ &\quad \Rightarrow t \in tags(b(new\_msg\_in)) \end{aligned} \tag{5}$$

The values of σ, the state of the system corresponding to `safe_FIFO`, can be defined by considering the signals `OK_write` and `OK_read`. The definition of these signals is itself based on the current number of enqueued messages, represented by `nbmsg`. Thus, σ can be characterized, using `nbmsg`, as follows: $\forall b \in [\![\texttt{safe\_FIFO}]\!]$ $\forall t \in tags(b)$,

$$\begin{aligned} b(\sigma)(t) = full &\quad \Leftrightarrow \quad b(nb\_msg)(t) = size \\ b(\sigma)(t) = empty &\quad \Leftrightarrow \quad b(nb\_msg)(t) = 0 \\ b(\sigma)(t) = none &\quad \Leftrightarrow \quad 0 < b(nb\_msg)(t) < size \end{aligned} \tag{6}$$

Since SIGALI does not address numerical properties, we must also find a Boolean encoding (or abstraction) for `nbmsg`.

4.2.2 Abstraction and verification by model-checking

As shown in FIG. 11, a n-FIFO queue can be represented by an automaton with $(n+1)$ states, where a state denotes the current number of messages in the queue. For the sake of simplicity, we consider a 2-FIFO queue since all possible relevant configurations can be addressed. So, the result remains valid for any bounded n-FIFO queue where $n > 2$.

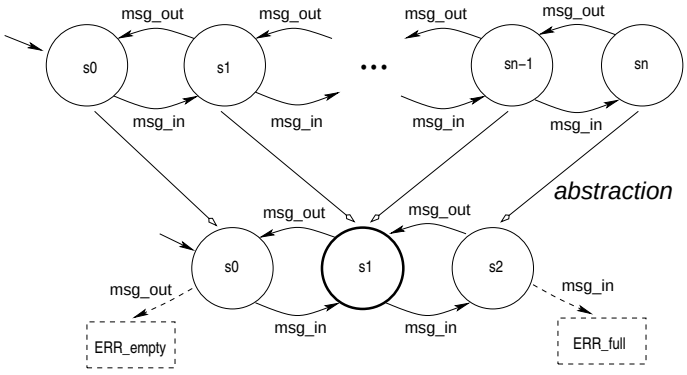


Fig. 11. Abstraction of a n-FIFO by a 2-FIFO with error states.

The automaton illustrated in FIG. 11 represents a behavior abstraction of a n-FIFO queue by a 2-FIFO queue. It relies on the behavioral isomorphism between both FIFO queues, which can be trivially observed. A state sk (represented by a circle) denotes that the queue contains k messages, i.e. $\forall k \in \{0,1,2\}$:

$$(nbmsg = k \Rightarrow sk = true) \wedge (nbmsg \neq k \Rightarrow sk = false)$$

The state $s0$ represents the initial state. Labels *msg_in* and *msg_out* are respectively write and read requests. Two special states (represented by rectangles) have also been added. They characterize "illegal" accesses to the queue: *ERR_empty* is reached on an attempt to read an empty queue, and *ERR_full* is reached when overwriting a full queue. They are also encoded by Boolean variables. Automata are very easy to specify in SIGNAL. For instance the state $s0$ and its associated transitions are defined as follows:

```
1    (|s0:= (true when prev_s1 when (^msg_out))
2        default (false when prev_s0 when
3        (msg_in ^+ msg_out)) default prev_s0
4     |prev_s0 := s0$1 init true |)
```

All the other states are specified in a similar way. It follows the definitions of signals `OK_write` and `OK_read` below:

```
1    (|OK_write:= false when (prev_err_full or
2                  prev_s2) default true
3     |OK_read:= false when (prev_err_empty or
4                  prev_s0) default true |)
```

The first equation means that a write request is not authorized when there are already two messages in the 2-FIFO queue (`prev_s2` is *true*), or when the queue has been overloaded previously (`prev_err_full` is *true*); otherwise it can be accepted. In a same way, the other statement specifies when a read request is legal.

The signals `s0`, `s1`, `s2`, `ERR_empty`, `ERR_full`, `OK_write` and `OK_read` are synchronized with `access_clock`.

The reachability of "error" states (i.e. safety properties) is concretely checked with SIGALI by considering the script given in [21].

To verify properties 3 and 4, we consider *observers*, represented by Boolean state variables. We have to show that these variables are always *true*. Let `obs4` and `obs3` denote respectively the observers for Properties 4 and 3.

- Property 4 is described as follows:
  - On a write request (denoted by the presence of `msg_in`), when the queue is either in $s0$ or $s1$; the signal `obs4` carries the value *true* if the message is actually written into the queue (i.e. `new_msg_in` is present), else `obs4` is *false*.
  - Otherwise, `obs4` keeps its previous value.

  The corresponding SIGNAL code is:

```
1    (|actual_w:= true
2       when(^new_msg_in) default false
3     |obs4:= actual_w when(z_s0 or z_s1)
4       when(^msg_in) default z_obs4
5     |z_obs4 := obs4 $ 1 init true |)
```

  The Boolean `actual_w` denotes the fact that a message is actually inserted in the queue.
- In a similar way, Property 3 is encoded by the following SIGNAL code:

```
1    (|actual_r := true
2       when(^msg_out) default false
3     |obs3 := actual_r when(z_s1 or z_s2)
4       when get_msg default z_obs3
5     |z_obs3 := obs3 $ 1 init true |)
```

Here also, all the new variables have the same clock as the signal `access_clock`. Then, the fairness properties can be checked as shown in [21].

4.2.3 Queueing order preservation

We have to prove that the `safe_FIFO` queue actually preserves the arrival order of enqueued messages. The idea consists in considering the flow of input messages in the call to `basic_FIFO` at line 13 in FIG. 9. Such a flow corresponds to that of signal `new_msg_in`. Indeed, only the values of this signal are taken into account and read afterwards. We therefore need to verify the following property:

Property 5 (order preservation):

- (E_1): *the order of values present in the input flow is preserved by the definition of the message queue*: $\forall b \in [\![\texttt{safe\_FIFO}]\!]$ $\forall v, v' \in \mathcal{V}$ s.t. $\exists t, t'$ with $v = b(new\_msg\_in)(t)$, $v' = b(new\_msg\_in)(t')$ and for i, i' the respective indices of v and v' in `queue`[2], the following holds: $t \leq t' \Rightarrow i \leq i'$.
- (E_2): *the order of values in the output flow is the same in the message queue*: $\forall b \in [\![\texttt{safe\_FIFO}]\!]$ $\forall v, v' \in \mathcal{V}$ t.q. $\exists t, t'$ with $v = b(msg\_out)(t)$, $v' = b(msg\_out)(t')$ and i, i' as above, the following holds: $i \leq i' \Rightarrow t \leq t'$. □

The order preservation property is specifically related to the properties of specific SIGNAL constructs used in the FIFO queue model. A proof of this property is in Appendix Section.

4.3 Use of `safe_FIFO` in architecture model

We have shown that the above `FW_System` model can be safely described with asynchronous communication mechanisms between its concurrent processes `Alarm_Manager` and `Alarm_Notifier`. Now, let us imagine a deployment of this system on a given architecture as illustrated in FIG. 3.

In this architecture, there are two processors *Processor 1* and *Processor 2* that communicate via a bus called *Bus*. The main feature of such an architecture that is taken into account in our

2. Here, the enqueued values are assumed to be inserted from right to left in the array defined with the `window` operator.

models is clock constraint. Each of the three components holds its own activation clock, which is different from those of others. Let us denote by κ_{p1}, κ_b and κ_{p2} the clocks corresponding respectively to *Processor 1*, *Bus* and *Processor 2*.

The bus model considered here is obtained from the previous `safe_FIFO` model on top of which a master clock is defined (see FIG. 12). This clock κ_b is represented by `^cnt` in the `Bus` process. It enables to define the instants at which incoming and outgoing transferred data are respectively accepted and delivered by the bus. This is specified through the signal `enable_in_out` at line 11. The static parameter `rw_per` denotes the period at which a read/write request is accepted by the bus. Its value belongs to the interval $0..k-1$ (note that `k` is also a static parameter in `Bus`).

```
1   process Bus =
2    { type msg_type; integer size;
3      msg_type def_msg; integer k, rw_per;}
4    ( ? integer msg_in;
5        event get_msg;
6      ! integer msg_out;
7        boolean OK_write, OK_read;
8    (| cnt:= ((k-1) when (zcnt = 0))
9               default (zcnt - 1)
10    | zcnt:= cnt $ init 0
11    | enable_in_out:= when(zcnt = rw_per)
12    | msg_in ^+ get_msg ^=
13          when enable_in_out
14    | (msg_out, OK_write, OK_read):=
15       safe_FIFO{msg_type,size,def_msg}
16                  (msg_in, get_msg)
17    |)
18  where
19     use safe_FIFO; integer cnt, zcnt;
20     event enable_in_out;
21  end;
```

Fig. 12. A bus model in SIGNAL.

On the other hand, we assume that the `Alarm_Manager` and `Alarm_Notifier` processes run at the frequency of their allocated processors: the master clock of `Alarm_Manager` is the same as κ_{p1} while the master clock of `Alarm_Notifier` is the same as κ_{p2}. So, processor constraints are implicitly taken into account by the `Alarm_Manager` and `Alarm_Notifier` models, which consequently remain unchanged.

Finally, the overall model of the deployed system is close to the one shown in FIG. 10, where `safe_FIFO` is replaced by `Bus`. Its clock analysis by the compiler exhibits the clock constraints that should be guaranteed by the program environment in order to perform functional simulation.

5 Analysis and Automatic Code Generation

5.1 Analysis of non functional properties

There are various ways to analyze non functional properties of SIGNAL models after their deployment on a given platform. For instance, performance evaluation is possible by using a quantitative temporal interpretation of SIGNAL programs [36], [23]. On the other hand, beyond the strict clock synchronization problems that are usually solved by almost all compilers of synchronous languages, more intricate synchronization issues can arise in the design of distributed systems if one would like to be able to widely explore architecture allocation possibilities. This requires sophisticated tools to address such issues.

5.1.1 Clock synchronizability issues

Affine clocks have been introduced in the SIGNAL clock calculus for this purpose [43], and more precisely in order to deal with synchronizability issues in a more "relaxed" way than usually in synchronous languages. Formally, an *affine transformation* of parameters (n, ϕ, d) applied to a clock κ_1 produces a clock κ_2 by inserting $(n-1)$ instants between any two successive instants of κ_1, and then counting on this fictional set of instants each d^{th} instant, starting with the ϕ^{th}. Clocks κ_1 and κ_2 are said to be in (n, ϕ, d)-affine relation, noted as $\kappa_1 \overset{(n,\phi,d)}{\rightarrow} \kappa_2$. The following trace depicts $\kappa_1 \overset{(3,1,4)}{\rightarrow} \kappa_2$, where "$t$" denotes the presence of a clock κ_i:

instants	0	1	2	3	4	5	6	7	8	9	...
κ_1:	t	$\perp$	$\perp$	t	$\perp$	$\perp$	t	$\perp$	$\perp$	t	...
κ_2:	$\perp$	t	$\perp$	$\perp$	$\perp$	t	$\perp$	$\perp$	$\perp$	t	...

In affine clock systems, two different clocks are said to be synchronizable if there is a dataflow preserving way to make them actually synchronous.

To show how such a notion can help to address non-trivial design decisions, let us consider the following allocation choices needed by a designer:

- to allow for a coherent communication protocol in this architecture, the following affine relations are assumed: $\kappa_{p1} \overset{(1,\phi_1,d_1)}{\rightarrow} \kappa_b$ and $\kappa_b \overset{(1,\phi_2,d_2)}{\rightarrow} \kappa_{p2}$;
- to satisfy some production-consumption rate which enables the `Alarm_Manager` process to confirm a certain number of alarms before their notification by the `Alarm_Notifier` process, an affine relation is assumed between their respective clocks clk_cnt1 and clk_cnt2: $clk\_cnt1 \overset{(1,\phi_3,d_3)}{\rightarrow} clk\_cnt2$.

While the first constraint is related to the architecture, the second one is purely functional.

Now, let us assume that only the clock clk_cnt1 of the `Alarm_Manager` process is identical to the clock of its associated processor, i.e. κ_{p1}. So, the main question is *which parameter values in the affine relations allow the designer to guarantee the synchronizability of the clocks clk_cnt2 and κ_{p2} with respect to the above architectural and functional constraints*.

The answer is given by solving the following system:

$$\begin{cases} \phi_1 + d_1\phi_2 = \phi_3 \\ d_1 d_2 = d_3 \end{cases} \quad (7)$$

The above simple example shows how to practically solve a typical synchronization problem in a polychronous design by using the SIGNAL compiler. We believe that a great advantage of this technique is that it enables to deal with large designs via adequate clock abstractions: one only needs to capture the relevant clock variables associated with system components and the synchronization relations between these clocks (from the full system specification). This will significantly reduce the size of the system model to be analyzed. Of course, the scalability limitation is that of our compiler [5].

More generally, formal verification tools, offering a support to the analysis of multi-clock systems, significantly help designers to adequately address several issues on the development of distributed embedded systems. An example is the bounded model-checking (BMC) technique, which has been applied to multi-clock systems in [24]. This method is scalable and allows one to check designs very fast thanks to SAT tools. It becomes worthy when, e.g., the compiler does not scale. However, its main inconvenient is that it only gives a partial guarantee

about proofs. So, BMC is clearly not sufficient alone but is a good complement to the other verification tools.

5.1.2 Applicability to the design of specific systems

Integrated Modular Avionics. Integrated Modular Avionics (IMA) [3] are typical distributed systems that can greatly benefit from the synchronous multi-clock modeling. In particular, the SIGNAL clock synchronizability analysis offers an interesting basis to deal with the very challenging issue of finding suitable resource allocation strategies in IMA architectures [7]. In such architectures [3]: *i)* time and memory resources are divided into *partitions*, which are associated with different functions at run-time; *ii)* partitions are composed of *processes*, representing the execution units that communicate within the same partition or with processes from other partitions. Partitions are scheduled according to a static cyclic policy while process scheduling is priority preemptive.

IMA-based implementation platforms assume the specific ARINC 629 Data Bus [1] and the 659 Backplane Data Bus [2]. Both buses adopt cyclic *time-triggered* message scheduling. In such a context, considering affine clocks can be significantly worth to designers to address the consistency of, on the one hand partition allocation to cyclic processors, and on the other hand cyclic processor schedules with ARINC 629 and 659 buses. This can be carried out on an IMA-based design of the FWS system, where processes are associated with partitions [23] that execute on different processors communicating via ARINC 629 and 659 buses. The system (7) enables to address partition allocation and communication constraints.

Further important implementation platforms for which the previous synchronizability analysis is very useful are those achieving communications with the *Avionics Full DupleX Switched Ethernet* (AFDX) technology [4] as in the Airbus A380. AFDX is very interesting because it is faster than its predecessors such as ARINC 629. It is based upon Ethernet and TCP/IP general principles. It is not time-triggered unlike the above two buses. Accordingly, with such a technology, dealing with the correct synchronization for the safe communication is more difficult. We believe that the polychronous model could be used to describe partial clock relations that capture the main synchronization points in an AFDX-based system. Then, the available analysis tools and techniques can be again applied to reason about the system behavior.

Large scale system-on-chip. The polychronous model is a good solution to deal with the shortcomings of GALS design frameworks in general, which adopt *ad hoc* methods where synchronous components are encapsulated with wrappers and communications are achieved by handshake. As illustrated in this paper, it allows one to uniformly model both aspects and formally validate the global system w.r.t. its correctness requirements. In addition, its vision of the synchrony-asynchrony link is suitable for large scale *system-on-chip* design [42].

5.2 Automatic code generation

After the non functional analysis of the deployed model of the FWS system, we can now consider an automatic modular code generation. Since, each "processor" is endochronous, the associated code can be generated independently from the other processors. The same holds for communications. This generation is necessarily achieved in the form of *clusters* [15]. In SIGNAL, a cluster denotes a group of statements that depend on the same sub-set of input data. This type of code generation is particularly interesting in that it enables an efficient execution for each processor. Indeed, the operating system supported by a processor is solicited less often because of the reduced number of commutations. The code corresponding to clusters is generated in various C, C++ or Java. In the current experimental implementation of the code generation process in POLYCHRONY, the obtained code is mainly intended for functional simulation of distributed applications.

6 DISCUSSIONS

All the design steps and analyses presented in our methodology are achieved within the POLYCHRONY environment [14], which provides the required tools, e.g. the SIGNAL compiler and the SIGALI model-checker. Along this presentation, we have stressed the advantages of our proposition for the reliable design of distributed embedded systems.

First, compared to mainstream implementation approaches that may rely for instance on C or Java programming, the synchronous programming offers well adapted specification concepts enabling a designer to safely describe systems. As an example, let us consider the endochrony property. Given a program p that satisfies such a property, p is able to read its required input data at the appropriate instants, based on its current state, from any asynchronous streams of input data. There are straightforward conditions in the compilation of SIGNAL programs, enabling to characterize this property of p: when the resulting overall clock hierarchy of p yields a tree of clocks. This property is not necessarily easy to achieve in mainstream approaches. In such approaches, test mechanisms for checking the availability of the input data required by a program must be defined. Endochrony will be therefore obtained by associating a given function in C or Java with an additional piece of code conferring this function the ability to decide alone which input data must be read during its execution. There are further benefits of synchronous programming over the use of synchronization mechanisms such as monitors in a language like Java. In synchronous programs, the fact that at any instant, an object, i.e. a signal, is written only once is guaranteed by construction. So, one does not need to consider some additional protection mechanism to enforce safe writes. As a result, the risk to get errors related to the usage of such delicate mechanisms is avoided. We can also mention the important role that play the formal tools and techniques provided by the synchronous technology for a trustworthy validation of designs.

From a methodological point of view, an interesting feature of our proposition is that it promotes correct-by-construction design: there is no need to (thoroughly) check the correctness of a system after design. This significantly reduces the validation efforts, which can take about 70% of the overall design cost. The abstract level at which the design is addressed (in the synchronous approach) enables to deal with the inherent complexity of distributed systems by leaving away the unnecessary details. It therefore favors a rapid exploration of different alternatives about system distribution so as to get relevant feedback for an adequate design choice.

However, in order to take advantage of all the above features of our approach in the design of real-life systems, the designer must have some knowledge of synchronous programming. In order to overcome this obstacle, there are currently significant efforts to provide a very advanced user-friendly interface for POLYCHRONY. The resulting new environment, referred to as SME, is an Eclipse plugin. It aims to be accessible to a wide

audience and is available online for free at the same address as POLYCHRONY. On the other hand, in the current status of our approach, some expertise in the SIGNAL clock calculus may be required to check the clock hierarchies, which serve to characterize the correctness of a system distribution. An important future improvement of our work is the automatic application of the distribution criteria (endo-isochrony) on any Signal model of a system so as to decide about its partitioning. Finding a solution to this issue will save the designer from analyzing possible complex clock hierarchies generated by the compiler for the distribution. Finally, for real systems, we can also mention another limitation of our approach, resulting from the simplifications considered in the architecture model. All physical components such as memory caches or hardware accelerators and their specific characteristics are not taken into account. Some abstractions of these components could be studied, and taken into account to ameliorate the distribution criteria.

7 CONCLUSIONS

In this paper, we presented how safety-critical distributed embedded systems are practically designed using the polychronous model, associated with the synchronous language SIGNAL. This model enables to describe systems with components that hold different activation clocks. We proposed a methodology that ensures a correct-by-construction implementation of these systems from high-level models in POLYCHRONY, the development framework of SIGNAL. This has been demonstrated on a case study from the avionic domain. The endochrony and endo-isochrony properties of SIGNAL specifications have been used as key notions to design the distribution of applications on execution platforms. These platforms contain asynchronous communication mechanisms, which have been also modeled in SIGNAL and proved to achieve message exchanges correctly. An important remark is that the formal verification techniques and tools available in POLYCHRONY are applicable at almost all design steps of our methodology in order to check the correctness of intermediate results. In particular, the analysis of non functional properties of the deployed system model has been addressed by dealing with clock synchronizability issues between the different multi-rate parts of the system. Finally, a distributed code can be automatically generated from this model.

A very interesting perspective to the presented work concerns the automatic application of the identified design criteria in SIGNAL to distribute a system. That is, given a system S, how could the compiler automatically use the endo-isochrony property to distinguish the different sub-parts $S_1, ..., S_n$ to be mapped on different processors such that $S = S_1 \mid ... \mid S_n$, and how could it select the suitable communication mechanisms from an existing library? It seems that the answer to these questions, in particular to the former, requires to analyze the combinatorics of the solution space of a system decomposition, which can be *a priori* large.

REFERENCES

[1] Airlines Electronic Engineering Committee (AEEC). ARINC 629: IMA Multi-transmitter Databus Parts 1-4, 1990.

[2] Airlines Electronic Engineering Committee (AEEC). ARINC 659: Backplane Data Bus, December 1993.

[3] Airlines Electronic Engineering Committee (AEEC). ARINC Specification 653: Avionics Application Software Standard Interface, Jan. 1997.

[4] Airlines Electronics Engineering Committee (AEEC). ARINC 664: Aircraft Data Network, Part 7: Avionics Full-Duplex Switched Ethernet (AFDX) Network, 2005.

[5] P. Amagbegnon, L. Besnard, and P. Le Guernic. Implementation of the data-flow synchronous language SIGNAL. In *In Conference on Programming Language Design and Implementation (PLDI'95)*, pages 163–173. ACM Press, 1995.

[6] P. Aubry. *Mises en œuvre distribuées de programmes synchrones.* PhD thesis, Université de Rennes I, IFSIC, France, 1997. (In french).

[7] N.C. Audsley and A.J. Wellings. Analysing APEX Applications. In *Real Time Systems Symp.(RTSS'96)*, Washington DC, USA, 1996.

[8] A. Benveniste. Safety critical embedded systems: the SACRES approach. In *proceedings of Formal techniques in Real-Time and Fault Tolerant Systems, FTRTFT'98 school, Lyngby, Denmark*, Sep. 1998.

[9] A. Benveniste, B. Caillaud, and P. Le Guernic. From synchrony to asynchrony. In *International Conference on Concurrency Theory (CONCUR'99)*, pages 162–177, London, UK, 1999. Springer-Verlag.

[10] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages twelve years later. *Proceedings of the IEEE*, 91(1):64–83, January 2003.

[11] A. Benveniste, P. Caspi, P. Le Guernic, H. Marchand, J-P. Talpin, and S. Tripakis. A protocol for loosely time-triggered architectures. In *Conf. on Embedded Software, EMSOFT'02, J. Sifakis and A. Sangiovanni-Vincentelli, Eds, LNCS vol 2491, Springer Verlag*, 2002.

[12] G. Berry and E. Sentovich. An implementation of constructive synchronous programs in POLIS. *Formal Methods in System Design*, 17(2):135–161, 2000.

[13] G. Berry and E. Sentovich. Multiclock esterel. In *11th Conference on Correct Hardware Design and Verification Methods (CHARME'01)*, pages 110–125, 2001.

[14] L. Besnard, T. Gautier, and P. Le Guernic. SIGNAL reference manual, 2007. www.irisa.fr/espresso/Polychrony.

[15] L. Besnard, T. Gautier, and J.-P. Talpin. Code generation strategies in the POLYCHRONY environment. Research Report RR-6894, INRIA, 2009.

[16] P. Caspi, A. Curic, A. Maignan, C. Sofronis, S. Tripakis, and P. Niebert. From simulink to SCADE/LUSTRE to TTA: a layered approach for distributed embedded applications. In *Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'03). San Diego, California, USA*, pages 153–162, 2003.

[17] D. Chapiro. *Globally Asynchronous Locally Synchronous Systems*. PhD thesis, Stanford University, 1984.

[18] B. Chéron. *Transformations syntaxiques de Programmes* SIGNAL. PhD thesis, Université de Rennes I, France, September 1991. (In french).

[19] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed systems (4th ed.): concepts and design*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005.

[20] M.J. Fischer, N.A. Lynch, and M.S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.

[21] A. Gamatié and T. Gautier. The SIGNAL approach to the design of system architectures. In *IEEE Conference on Engineering of Computer-Based Systems (ECBS'03), Huntsville USA*, April 2003.

[22] A. Gamatié, T. Gautier, and P. Le Guernic. Towards static ananlysis of SIGNAL programs using interval techniques. In *Synchronous Languages, Applications, and Programming (SLAP'06)*, March 2006.

[23] A. Gamatié, T. Gautier, P. Le Guernic, and J.-P. Talpin. Polychronous design of embedded real-time applications. *ACM Transaction on Software Engineering Methodology*, 16(2):9, 2007.

[24] M. K. Ganai and A. Gupta. Efficient BMC for Multi-Clock Systems with Clocked Specifications. In *Conference on Asia South Pacific design automation*, pages 310–315, 2007.

[25] T. Gautier and P. Le Guernic. Code generation in the SACRES project. In *Safety-critical Systems Symp. (SSS'99)*. Huntingdon, UK, Feb. 1999.

[26] A. Girault. *Sur la répartition de programmes synchrones.* PhD thesis, Institut National Polytech. de Grenoble, France, 1994. (In french).

[27] A. Girault. A survey of automatic distribution method for synchronous programs. In F. Maraninchi, M. Pouzet, and V. Roy, editors, *Synchronous Languages, Applications and Programs (SLAP'05)*, ENTCS, Edinburgh, UK, April 2005. Elsevier Science.

[28] A. Girault and C. Ménier. Automatic production of globally asynchronous locally synchronous systems. In A. Sangiovanni-Vincentelli and J. Sifakis, editors, *Workshop on Embedded Software (EMSOFT'02)*, volume 2491 of *LNCS*, pages 266–281. Springer-Verlag, 2002.

[29] A. Girault, X. Nicollin, and M. Pouzet. Automatic rate desynchronization of embedded reactive programs. *ACM Transaction on Embedded Computing Systems*, 5(3):687–717, August 2006.
[30] T. Grandpierre. *Modélisation d'architectures parallèles hétérogènes pour la génération automatique d'exécutifs distribués temps réel optimisés.* PhD thesis, Université de Paris-Sud, Nov. 2000. (In french).
[31] T. Grandpierre and Y. Sorel. From algorithm and architecture specifications to automatic generation of distributed real-time executives: a seamless flow of graphs transformations. In *MEMOCODE2003, Formal Methods and Models for Codesign Conference, Mont Saint-Michel, France*, June 2003.
[32] R. Gupta, S. Pande, K. Psarris, and V. Sarkar. Compilation techniques for parallel systems. *Parallel Computing*, 25(13–14):1741–1783, 1999.
[33] O. Hainque. *Etude d'un environnement d'exécution temps-réel, distribué et tolérant aux pannes pour le modèle synchrone.* PhD thesis, École Nationale Supérieure des Télécom. - Paris, 2000. (In french).
[34] N. Halbwachs and S. Baghdadi. Synchronous modeling of asynchronous systems. In *EMSOFT'02*, Grenoble, October 2002. LNCS 2491, Springer Verlag.
[35] N. Halbwachs and L. Mandel. Simulation and verification of asynchronous systems by means of a synchronous model. In *Sixth International Conference on Application of Concurrency to System Design, ACSD 2006*, Turku, Finland, June 2006.
[36] A. Kountouris and P. Le Guernic. Profiling of SIGNAL programs and its application in the timing evaluation of design implementations. In *Proceedings of the IEE Colloq. on HW-SW Cosynthesis for Reconfigurable Systems*, pages 6/1–6/9, Bristol, UK, February 1996.
[37] P. Le Guernic, J.-P. Talpin, and J.-C. Le Lann. Polychrony for System Design. *Journal for Circuits, Systems and Computers*, 12(3):261–304, April 2003.
[38] N. Lopez, M. Simonot, and V. Donzeau-Gouge. A methodological process for the design of a large system: two industrial case-studies. *Electronic Notes in Theoretical Computer Science*, 66(2), 2002.
[39] O. Maffeïs. *Ordonnancements de graphes de flots synchrones : application à la mise en œuvre de* SIGNAL. PhD thesis, Université de Rennes I, IFSIC, France, January 1993. (In french).
[40] H. Marchand, P. Bournai, M. Le Borgne, and P. Le Guernic. Synthesis of discrete-event controllers based on the SIGNAL environment. In *Discrete Event Dynamic System: Theory and Applications, 10(4)*, pages 325–346, October 2000.
[41] M.R. Mousavi, P. Le Guernic, J.-P. Talpin, S.K. Shukla, and T. Basten. Modeling and validating globally asynchronous design in synchronous frameworks. In *DATE*, pages 384–389, 2004.
[42] J. Muttersbach, T. Villiger, H. Kaeslin, N. Felber, and W. Fichtner. Globally-Asynchronous Locally-Synchronous Architectures to Simplify the Design of On-chip Systems. In *12th IEEE International ASIC/SOC Conference*, Washington DC, USA, 1999.
[43] I.M. Smarandache, T. Gautier, and P. Le Guernic. Validation of mixed SIGNAL-ALPHA real-time systems through affine calculus on clock synchronisation constraints. In *World Congress on Formal Methods (2)*, pages 1364–1383, 1999.
[44] A. Tanenbaum and M. van Steen. *Distributed Systems: Principles and Paradigms (2nd ed.)*. Prentice Hall, 2007.
[45] P. Veríssimo. On the role of time in distributed systems. In *6th IEEE Workshop on Future Trends of Distributed Computer Systems (FTDCS'97), 29-31 October 1997, Tunis, Tunisia*, pages 316–323. IEEE Computer Society, 1997.

APPENDIX
PROOF OF PROPERTY 5 (ORDER PRESERVATION)

Proof of (E_1)**.** The proof of this property mainly relies on the semantics of the `window` operator (see Section 4.1.1). As a matter of fact, the message queue represented by the signal `queue` is defined by applying this operator to the signal `msg_in` (see equation 10 in FIG. 8). The flow associated with this signal contains the input messages of the queue. The semantics of this operator is such that every reading of value on the input flow leads to a sliding on the left of the values already present in the defined array. The new value is therefore inserted on the right. The following trace illustrates such a scenario on a 2-sized queue. The symbol "$\star$" designates a given initialization value.

`msg_in` :	$\perp$	1	2	$\perp$	3	4	...
`queue` :	$[\star,\star]$	$[\star,1]$	$[1,2]$	$[1,2]$	$[2,3]$	$[3,4]$	...

For the equation: `queue := new_msg_in window size`, the semantics of the *sliding window* yields (for the sake of simplicity, initial values are omitted): $\forall t \geq 0$,

$$(t+i \geq size) \Rightarrow (queue_t[i] = new\_msg\_in_{t-size+i+1}).$$

Thanks to this semantics of the `window` operator, the values that are taken into account in the flow associated with the input signal `msg_in` are put in `queue` according to their occurrence order. Thus, the property (E_1) holds by definition.

Proof of (E_2)**.** The output flow is determined by the equation defined at lines 11 and 12 in FIG. 8. In `safe_FIFO`, only the second argument of the `default` operator has to be considered since the first one is forbidden by Property 2. On a read request, we denote by i the index of the message retrieved from the array `queue`. The value of i is given by the difference between the maximal size (i.e. `size`) of the message queue and the previous number of messages, `prev_nbmsg` of the queue: $\forall t \geq 0 \quad i_t = fifo\_size - prev\_nbmsg_t$.

Let us consider the following invariant: *at any logical instant t, the index i_t indicates the oldest message enqueued when the message queue is not empty*. We need to check that this invariant is always preserved after any access to the queue.

1) Case 1: *read at an instant k.* When a message is read at an instant k, the signal `nbmsg` is decremented by one (equation at lines 5 and 6 in FIG. 8), i.e.: $nbmsg_k = prev\_nbmsg_k - 1$. At instant $k+1$, we have: $prev\_nbmsg_{k+1} = nbmsg_k$. Thus,

$$\begin{aligned} i_{k+1} &= fifo\_size - prev\_nbmsg_{k+1} \\ &= fifo\_size - nbmsg_k \\ &= fifo\_size - (prev\_nbmsg_k - 1) \\ &= (fifo\_size - prev\_nbmsg_k) + 1 \\ &= i_k + 1 \end{aligned}$$

On the other hand, on each read request, the array represented by the signal `queue` remains unmodified. As a result, the invariant is preserved since the oldest element in the message queue becomes the one with index i_{k+1} in the array.

2) Case 2: *write at an instant k.* When a message is effectively written in the message queue at an instant k, the signal `nbmsg` is incremented by one (equation at lines 5 and 6 in FIG. 8), i.e.: $nbmsg_k = prev\_nbmsg_k + 1$. At instant $k+1$, we have: $prev\_nbmsg_{k+1} = nbmsg_k$. Thus, similarly to the previous case, we obtain that: $i_{k+1} = i_k - 1$.

Contrarily to Case 1, here the signal `queue` is modified. As a matter of fact, the sliding window on the signal `new_msg_in` will take into account the occurrence of this signal at instant k. The elements that are already present in `queue` are shifted according to the semantics of the `window` operator (i.e. a shift of one cell towards lower indices of the array and insertion of the new message in the cell with the highest index value). Hence, the index i_{k+1} always corresponds to the same message as the previous instant. So, the invariant is preserved.